Batteries not included

Josiah Hester and Jacob Sorber

Preprint of ACM XRDS Student Magazine Feature (September 2019)

For those of you just emerging from decades of cryosleep, this is where things stand. We don't have colonies on Mars or flying cars. But we do have (a lot of) smarter, prettier phones; we have rebranded "ubiquitous computing" and "smart dust"—now called the Internet of Things (IoT); and we have a battery problem.

Batteries are at once the best friend and the worst enemy of the IoT—having both fueled myriad advances over the last two decades in mobile computing, and yet severely limited the scale, scope, and consequences of our deployments. Batteryless and energy harvesting computing is a recent, exciting alternative promising decade-long maintenance-free deployments, but also requiring us to fundamentally change how we design very small computing systems.

For decades, batteries have supported our mobile computing advances, enabling countless new applications and giving our devices stable power for hours or days—long enough to be useful between charges. But batteries also add size, weight, and cost; present safety and environmental hazards; require special charging and protection circuitry; and wear out after 3-5 years, even when regularly and carefully recharged. Replacing and disposing of trillions of batteries every few years is inconvenient, expensive, and irresponsible.

So, what should we do? We would love better batteries, but battery technologies have traditionally improved slowly. We wish physicists and chemists the best of luck, but we're not willing to wait a decade or two. Instead, we have decided to build mobile devices that don't need batteries. Devices that survive using harvested energy, store that energy in small capacitors, and work opportunistically maintenance-free for decades. Devices that make the most of energy when it is available and get useful work done in the face of frequent power failures. If successful, batteryless computing will make it possible to deploy maintenance-free untethered devices that communicate and gather data in hard-to-reach places.

It sounds great, but it's complicated. With stable battery power, operating systems, software libraries, and applications have moved from desktops to laptops and phones with only minor incremental changes. Inside, smartphones still look a lot like little laptops, but unreliably powered batteryless computing devices are fundamentally different from their ancestors. Energy harvesting is inconsistent,

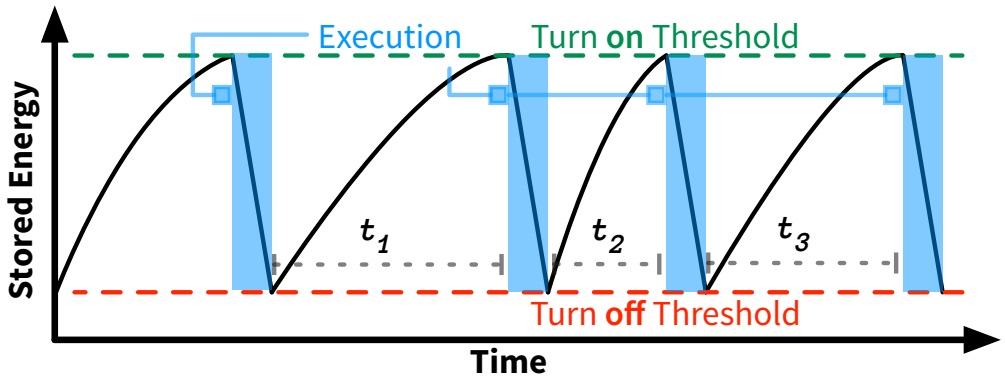

Figure 1: A typical intermittent execution pattern for an energy-harvesting batteryless device.

energy storage is scarce, power failures are inevitable, and execution is often intermittent, as shown in Figure 1.

Intermittence changes things. Power failures are no longer rare exceptions. They are the rule. The developer is trying to get useful work done, in spite of these interruptions. Simple concerns like keeping time, making forward progress, and keeping data structures consistent are challenging. Power fluctuations are difficult to predict, and outages can last for seconds or hours. A device's run-time behavior determines how much energy it will harvest—meaning application or system decisions like when to schedule tasks, how to use low power modes, and when to communicate will impact both how much energy the device will use and how much it will harvest. In the face of this extra complexity, application developers often struggle to predict how their programs will function in the wild and anticipate the problems that are likely to occur.

Our goal is to help bring batteryless intermittent computing out of its infancy. For decades, we've built tools, languages, algorithms, and hardware platforms that assume power is reliable and outages are rare. When it comes to developer support for batteryless systems, we're just getting started. We envision a future where amateurs can confidently program, test, and deploy batteryless intermittent applications. Where researchers and application designers have time-tested hardware platforms, software techniques, tools, runtime systems, and programming abstractions that allow them to focus on the logic of their programs and the goals of their applications, instead of worrying about the next power outage. In the last few years, we have seen a lot of progress—a good start—but this article is really a call for more help to develop the next generation of platforms and tools, languages and runtimes, and intermittent networks that will fuel the batteryless revolution.

Platforms and Tools

New research communities thrive and grow when they have usable platforms and tools that simplify development and exploration.

Hardware. A batteryless application's success often depends on hardware design decisions or tuning. We have come a long way since the early days of computational RFIDs [1]. CRFIDs are simple and completely dependent on being close to an RFID reader, and this limits the applications they can support. As we have explored other energy sources (light, vibration, motion, and thermal gradients) and the applications they enable, our hardware is now more sophisticated. We can federate a device's energy storage [2]—providing separate prioritized energy storage capacitors for processing and individual peripherals—to make energy harvesting more efficient and programming simpler.

Small capacitors charge more quickly for short bursts of low energy peripheral usage (for example sensing acceleration). Larger capacitors charge slowly, but support longer compute tasks and power-hungry peripherals like a radio or gesture sensor. With federated energy, a device can support more peripherals without sacrificing availability. By making this energy federation dynamic and at runtime, platforms like Flicker [3] and Capybara [4] let programmers decide on the fly what is important and what to save energy for.

Developing effective batteryless hardware has traditionally required an electrical engineering background. Developers built custom hardware and specialized highly tuned software. The learning curve is still steep, but we're trying to make it easier with Flicker—a flexible, modular hardware platform inspired by Arduino shields for prototyping batteryless applications. Flicker provides a general peripheral interface that simplifies software development and makes it easier to extend support for other hardware peripherals.

With Flicker, a designer can evaluate the hardware requirements for the application they want to build (for example, environmental monitoring), choose the Flicker peripherals they want to use (for example, a radio transceiver, moisture sensor, and humidity sensor), then plug in those modules and an energy harvester into the main Flicker board to immediately start writing code. By shortening design time from weeks to minutes by enabling rapid prototyping, Flicker broadens the reach of batteryless sensing, lowering the barrier to entry, and reduces the startup costs.

Timekeeping. Time is one of the most important (and taken for granted) resources in any computing system. Accurate and available timekeeping underlies most

system services and security features of modern applications. One of the challenges for batteryless computing is keeping track of time across power failures. We've learned to keep time, even in the face of power outages. Once the device dies, and it will, all the volatile memory, and all the internal clocks are reset. This means any previous timestamps on data have no reference point since the internal clock will start back at zero. To fix this, we can use "electronic hourglasses" to keep time while the batteryless computing device is in failure [5]. These electronic hourglasses are just a capacitive element that is "turned over" every time the device reboots. After boot, our batteryless computer polls the hourglass to see how much time has elapsed since it died, then resets and maintains the hourglass until the next reboot, allowing persistence of time through power failures.

Emulation/experimentation. We have found ways to debug intermittent programs and compare their performance in a rigorous way. With traditional computing platforms, comparing different algorithms is straightforward. We can measure the speed, energy consumption, or another metric, and for the same workload the results will be comparable. When we compare energy harvesting systems the intrinsic attributes of energy harvesting complicate matters.

The amount of power that you harvest from a solar panel (or other harvester) depends on how much energy is already stored. So, the power depends on the supply voltage. If two programs consume different amounts of energy, those two programs will harvest different amounts of energy. So your harvesting efficiency depends on your program behavior. When light hits a solar panel, it produces a function that describes how much harvested current will flow at each capacitor voltage. We can model or record these functions and emulate them using Ekho [6], which gives us repeatable experimentation and accurate emulation of energy harvesting conditions. Even better, digitized representations of energy harvesting conditions in South Carolina can be emailed to researchers in Chicago for testing and the results will be the same.

Languages, Runtimes, and Operating Systems

Understanding the operation of a program through intermittent power failures is a cognitive burden on the developer. Intermittent operation is not just difficult for the machine, the humans programming them also have to wrap their heads around complex bugs, stale data, and timing errors. Execution on a batteryless device consists of harvesting energy for a relatively long time, then a short execution, followed by a power failure once the supply falls below the processors' operating voltage. Once this power failure happens, the clock is set back to zero, all the contents of RAM are lost, and registers and stack are reset. Eventually, once

enough energy has been harvested, the program will restart back at main. The problem is this power failure could occur between any two lines of code. Despite advances in tooling and hardware, developers still find programming batteryless devices very challenging and nonintuitive. Composing programs for batteryless, intermittently powered devices still requires in-depth knowledge of hardware and energy harvesting behaviors.

Checkpointing. We expect a program to execute line by line, with instructions quickly following one after another. For batteryless sensors that fail 10 times a second, our execution will be hindered by numerous interruptions and restarts. The first approach we tried was to save the program state right before a power failure, then restore that state once the device turns back on. Mementos [7], the first runtime checkpointing tool for batteryless devices, used a compiler pass to insert energy checks and checkpoint logic throughout the program code. After a power failure, the most recent checkpoint was restored, and program execution continued on normally. Many researchers have since improved on Mementos, making checkpoints smaller, avoiding unnecessary checkpoints, and saving program state more strategically—all statically or dynamically checkpointing some subset of program state to make sure a program is always making forward progress.

Task-based languages. Checkpointing based approaches are not the only way to make intermittent programming easier and more efficient. Checkpointing approaches sacrifice overhead (in terms of memory impact and speed) to simplify conversion of code to the intermittent application space. An alternative approach is to wrap functionality into discrete "tasks" that are linked together to specify control flow and data flow. Instead of checkpointing, this model asks programmers to bear part of the work of controlling for intermittency by organizing or rewriting their program into atomic "tasks" and specifying how the "task graph" behaves. Task-based languages incur much lower overhead because they save less data. Data dependencies are explicit and defined at compile time, allowing a task-based system to focus more energy on scheduling high-priority tasks and maximizing useful work in spite of power failures.

Timing failures. No matter the approach, fixating only on forward progress can be a mistake. Continuing where you left off after a power failure is not always the best thing for an application. After a long power failure, old sensor data may no longer be relevant (for example, motion readings in an activity detection application). Computing on stale data is a waste of computing time and precious energy. The task-based language Mayfly [8] is a response to this. Mayfly asks programmers to define a task graph, linking tasks such as "sense temperature" with "compute

average" and "send data." These links are then annotated with soft timing requirements, for example, requiring that any data from the sensor must be less than 10 seconds old to send over the radio. Mayfly captures the relationship between time and data that checkpointing and forward-progress focused systems do not; sometimes forward progress is not enough, because time passes during power failures.

Intermittent Networks

Today batteryless networks don't exist, because we don't know how to effectively coordinate and synchronize multiple intermittently-powered devices. Batteryless devices may have only enough energy stored up to send or receive a few packets, long network conversations and long-distance transmissions may be infeasible. A node may not have the energy to communicate at a particular time, and a node's sense of time will also often be too inaccurate for low-power protocols, like Bluetooth low energy (BLE), that rely on tight time synchronization.

The future of intermittent devices hinges on how well they can communicate, and in order to teach them to communicate we need to rethink some fundamental assumptions about how low-power networks work.

Listening is expensive, so low-power networks listen periodically or on a set schedule (TDMA) to avoid listening to the channel as much as possible. With frequent power outages and imprecise timekeeping, intermittent devices have trouble sticking to a precise schedule, and they can't afford the energy that BLE spends on neighbor discovery. When batteryless devices communicate with active radios, they simply transmit packets blindly, hoping that an always-listening base station will receive them. Spray-and-pray networking is simple but underwhelming, limiting us to one-hop topologies, power-hungry base stations, and possible collision problems as the number of transmitting nodes grows.

On the other hand, with low-power signal-detection circuits and wake-up radios, best-effort synchronization, and protocols that tolerate and adapt to timing errors we hope for more flexible and efficient batteryless networks.

Backscatter-based communication [9] used by RFID tags, is a promising alternative —using a radio frequency source to provide both power and communication (and synchronization). Communication requires less power and synchronization is straightforward, but backscatter requires installed infrastructure that may not be universally available. Also communication range is limited and the device needs to be close to a transmitter (especially if we want the radio frequency signal to power

the device). It is also unclear how backscatter techniques will scale to large numbers of devices as communication ranges increase.

What's Next?

Hopefully, you're now excited about batteryless intermittent computing. We hope you're wondering how you can get involved. We've already mentioned several key research areas, but let's finish with a few of the open research topics that we're excited about.

Reprogramming. While we're designing technologies that can be deployed for decades, we would be naive to think these devices won't, at times, need a software update. Reprogramming and retasking devices in the field is challenging even with reliable power, and none of the solutions that exist are suitable for the energy and network constraints that are typical of batteryless devices.

Learning, inference, and adaptation. Batteryless devices will experience a wide range of energy and environmental conditions, and they will be tasked with interpreting complicated signals and making sense out of the world around them. Will batteryless intermittent devices take advantage of the recent machine learning wave? The standard approach of sending large amounts of raw data to the cloud for processing will likely not be practical. How much can we learn and infer on such scant resources? It's difficult for developers to predict what conditions their applications will meet when deployed. Can our devices learn from their mistakes in the field, and learn how to effectively adapt to meet their changing constraints?

Sharing the load. Leaving batteries behind significantly lowers the cost of individual devices, leading to more dense deployment, and potentially leading to the trillion device Internet-of-Things. As these devices become ever more closely integrated into the physical world, and networked with each other, opportunities for these devices to work together become more apparent. Challenges remain in developing new programming models that share the computational load across this sensing fabric This becomes even more difficult as the heterogeneity of devices and applications increases. Abstractions that gracefully share hardware and software resources across the swarm, but do not overburden the developer will need to be created.

Securing a trillion devices. Battery-free devices open up exciting new application domains in health, space, and infrastructure monitoring because of the promise of long maintenance free lifetimes. However, these applications are sensitive in terms of privacy and security. Wearable body health networks must safeguard a patient's

data at all times, but with limited energy. Infrastructure monitoring provides the sensor data that helps building managers make decisions; if these sensors are wrong or compromised, poor decision-making follows. It is not clear what are the best strategies for securing the smallest and most energy-poor computational devices that are easily physically accessed. Challenges exist in building secure clocks, attestation protocols, sensor spoofing resilience, and defense against energy-based denial of service attacks. Providing security through intermittent execution, however, will enable scale-up into the most impactful applications and research domains.

So, welcome to 2019. There's work to be done. We hope you'll help us solve some of these issues on our way to the next trillion devices.

References

[1] A. N. Parks, A. P. Sample, Y. Zhao, and J. R. Smith. A wireless sensing platform utilizing ambient RF energy. In Proceedings of the 2013 IEEE Radio and Wireless Symposium. IEEE, 2013, 331–333.

[2] Hester J., Sitanayah L., and Sorber J. Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors. In Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys '15]. ACM, 2015, 5–16.

[3] Hester J. and Sorber J. Flicker: Rapid prototyping for the batteryless internet-of-things. In Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17]. ACM, 2017, 1–13.

[4] Colin A., Ruppel E., and Lucia B. A reconfigurable energy storage architecture for energy-harvesting devices. In Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18). ACM, 2018, 767–781.

[5] Hester J, Tobias N., Rahmati A., Sitanayah L., Holcomb D., Fu K., Burleson W.P., and Sorber J. Persistent clocks for batteryless sensing devices. ACM Trans. Embed. Comput. Syst. 15, 4 (2016).

[6] Hester J., Scott T., and Sorber J. Ekho: realistic and repeatable experimentation for tiny energy-harvesting sensors. In Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys '14]. ACM, 2014, 330–331.

[7] Ransford B., Sorber J., and Fu K. Mementos: System support for long-running computation on RFID-scale devices. In Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI). ACM, 2011, 159–170.

[8] Hester J., Storer K., and Sorber J. Timely execution on intermittently powered batteryless sensors. In Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17). ACM, New York, 2017.

[9] Varshney A., Harms O., Pérez-Penichet C., Rohner C., Hermans F., and Voigt T. LoRea: A backscatter architecture that achieves a long communication range. In Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17). ACM, New York, 2017.

Author Biographies

Josiah Hester | josiah@northwestern.edu
Assistant Professor of Computer Engineering
Department of Electrical and Computer Engineering
Department of Computer Science
Northwestern University
josiahhester.com

Jacob Sorber | jsorber@clemson.edu
Associate Professor of Computer Science
School of Computing
Clemson University
www.jacobsorber.com.